Improved State Space Reductions for LTL Model Checking of C & C++ Programs

P. Ročkai, J. Barnat, L. Brim

Faculty of Informatics
Masaryk University Brno

May the 14th, 2013

DIVINE

- **DIVINE** is an explicit-state LTL model checker,
- which does C and C++ verification via LLVM,
- and does so in parallel, and possibly in a cluster.

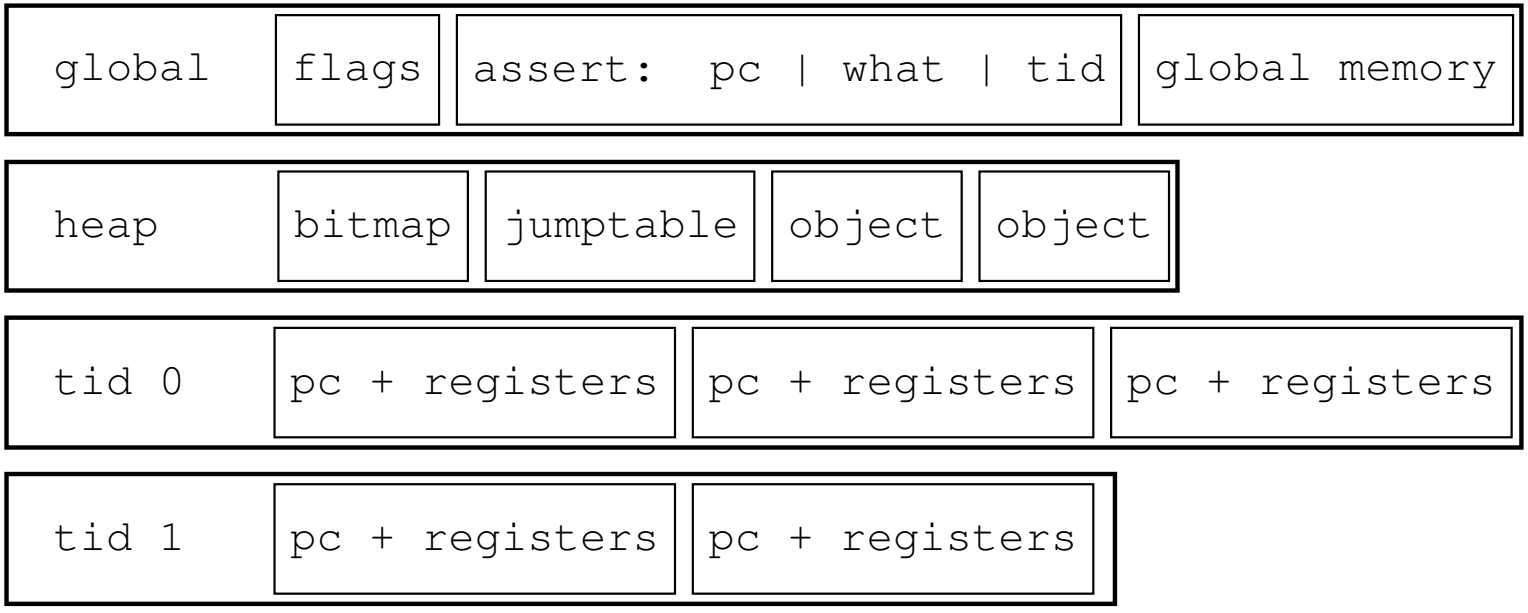
(Yes, it is awesome. The mileage is great and your wife will *love* the color.)

LLVM

- Compilation & optimisation infrastructure,
- with a workable intermediate representation & library.
- Low-level, resembling a CPU instruction set,
- widely deployed in practice.

- C & C++ Frontends: Clang, dragonegg (GCC)

What is a Program State?



Stored as a flat vector of bytes in memory, sent across network, held in a hash table.

Program State Spaces

- The state space of a parallel program is **HUGE** ...
- Very simple model, peterson on 2 threads, no reductions:
- doesn't finish in 16GB of memory. (Ouch.)

Opportunities

- Excessive interleaving:
  - τ+reduction,
  - `store` visibility,
  - atomic sections in library routines.
- Redundant heap configurations:
  - heap symmetry reduction.

Counter-Examples vs Reductions

- A counter-example in 1000 steps is (mostly) useless.
- **Better** reductions yield **shorter** counter-examples.

That ubiquitous peterson model again:

- no reductions, almost 200 steps in a counter-example;
- fully reduced: 12 steps

How to lose 20 pounds in 2 weeks?

... why, just use magic.

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
```

generated: $\emptyset$

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
```

generated: $\emptyset$

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
```

generated: $\emptyset$

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
```

generated: $\emptyset$

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
```

generated: $\emptyset$

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
```

generated: $\emptyset$

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
generated: %7+3: @i = 1
```

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
generated: %7+3: @i = 1
```

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
generated: %7+3: @i = 1
```

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
generated: %7+3: @i = 1
```

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
generated: %7+3: @i = 1
```

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
generated: %7+3: @i = 1
```

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
```

generated: `%7+3: @i = 1`
`%7+2: @i = 2`

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
```

generated: `%7+3: @i = 1`
`%7+2: @i = 2`

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
```

generated: `%7+3: @i = 1`
`%7+2: @i = 2`

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
```

generated: %7+3: @i = 1
%7+2: @i = 2

$\tau+$reduction

- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
generated: %7+3: @i = 1
           %7+2: @i = 2
```

$\tau+$reduction

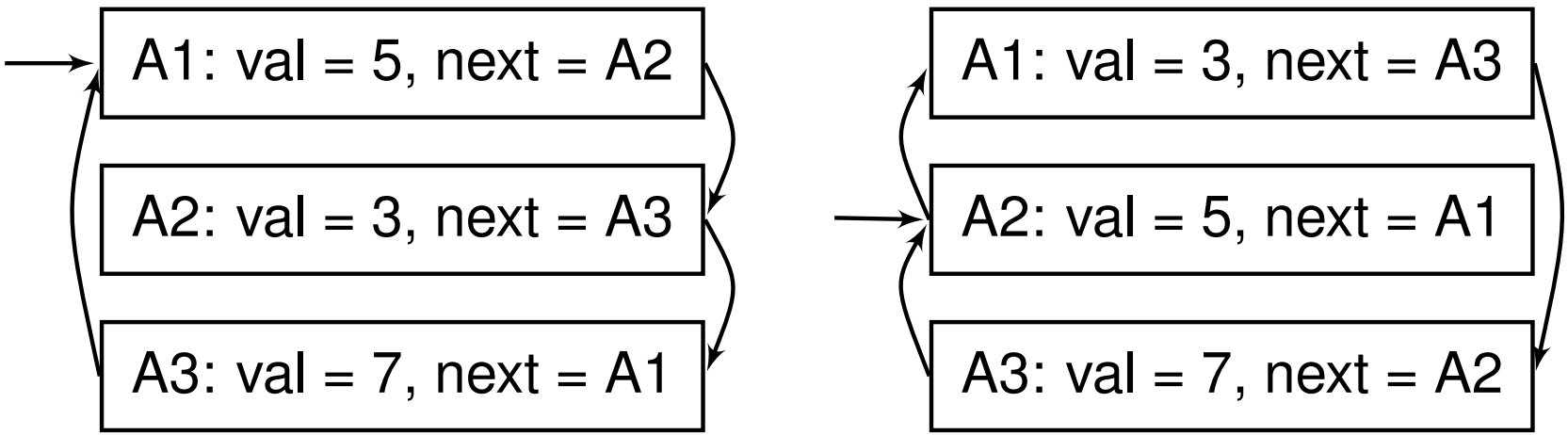
- Combination of path reduction & partial order reduction.
- Based on τ-reduction, adding a dynamic revisit check.

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
```

generated: `%7+3: @i = 1`
`%7+2: @i = 2`
`%4+3: @i = 2`

Heap Symmetry

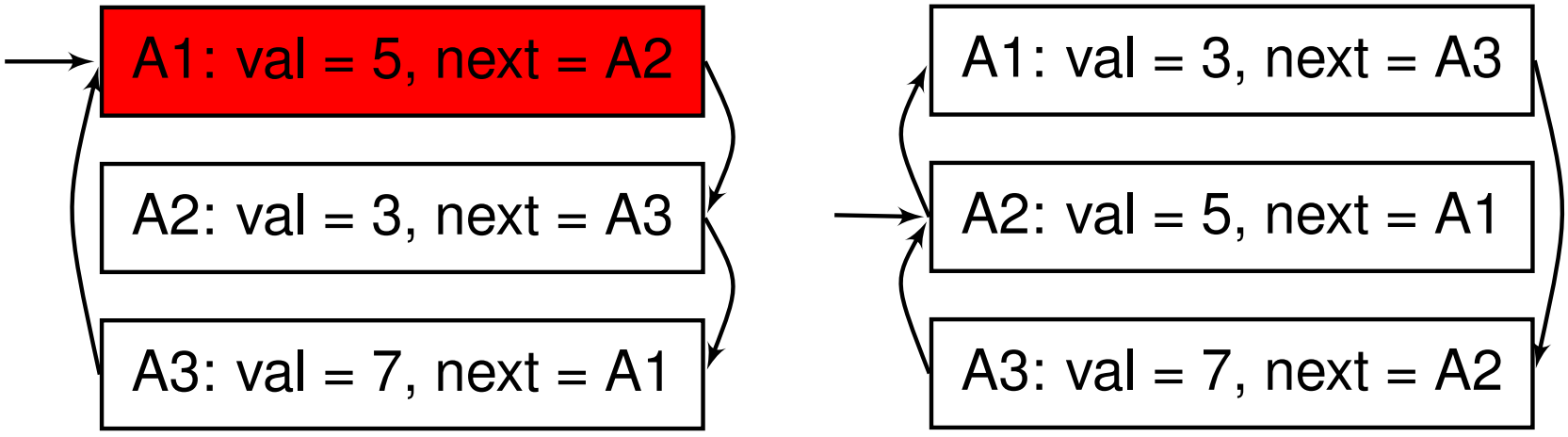
- Sometimes, different states are actually the same.
- $\Rightarrow$ they are symmetric, and we only want one.



- This happens a lot when allocations are interleaved.
- Sort topologically from fixed roots. Keep track of all pointers, instrument code. Get a canonic layout. Magic.

Heap Symmetry

- Sometimes, different states are actually the same.
- $\Rightarrow$ they are symmetric, and we only want one.

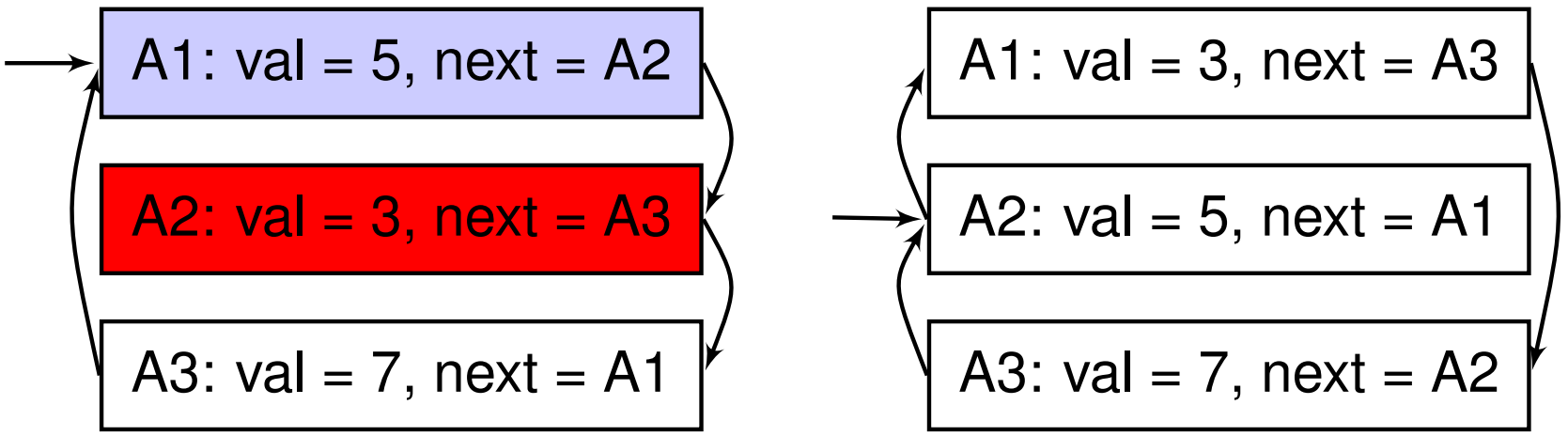


$\rightarrow 5$

- This happens a lot when allocations are interleaved.
- Sort topologically from fixed roots. Keep track of all pointers, instrument code. Get a canonic layout. Magic.

Heap Symmetry

- Sometimes, different states are actually the same.
- $\Rightarrow$ they are symmetric, and we only want one.

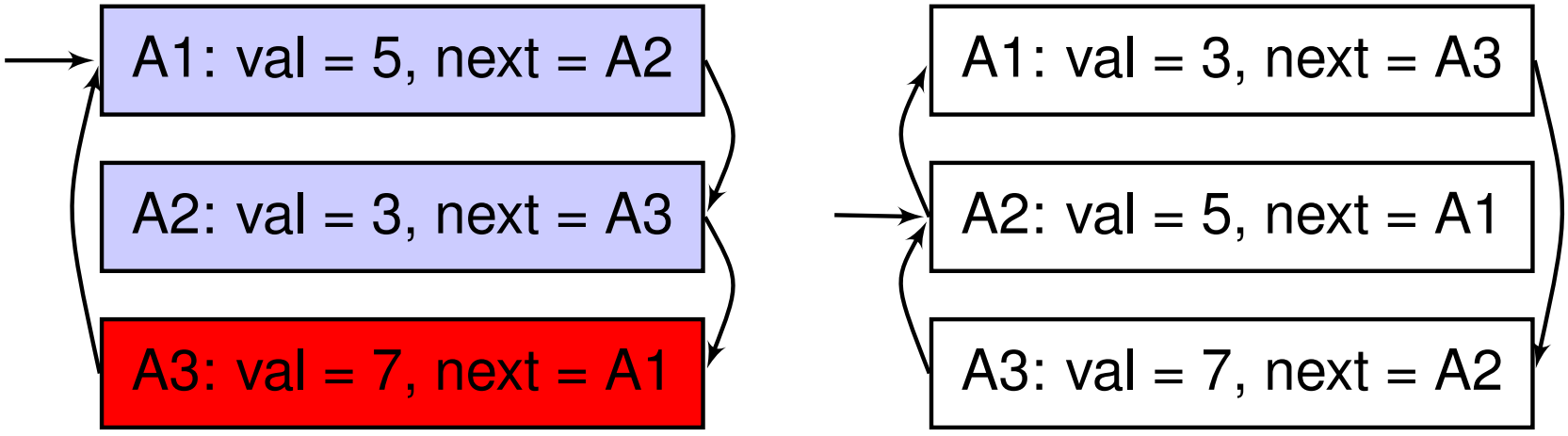


$\rightarrow 5 \rightarrow 3$

- This happens a lot when allocations are interleaved.
- Sort topologically from fixed roots. Keep track of all pointers, instrument code. Get a canonic layout. Magic.

Heap Symmetry

- Sometimes, different states are actually the same.
- $\Rightarrow$ they are symmetric, and we only want one.

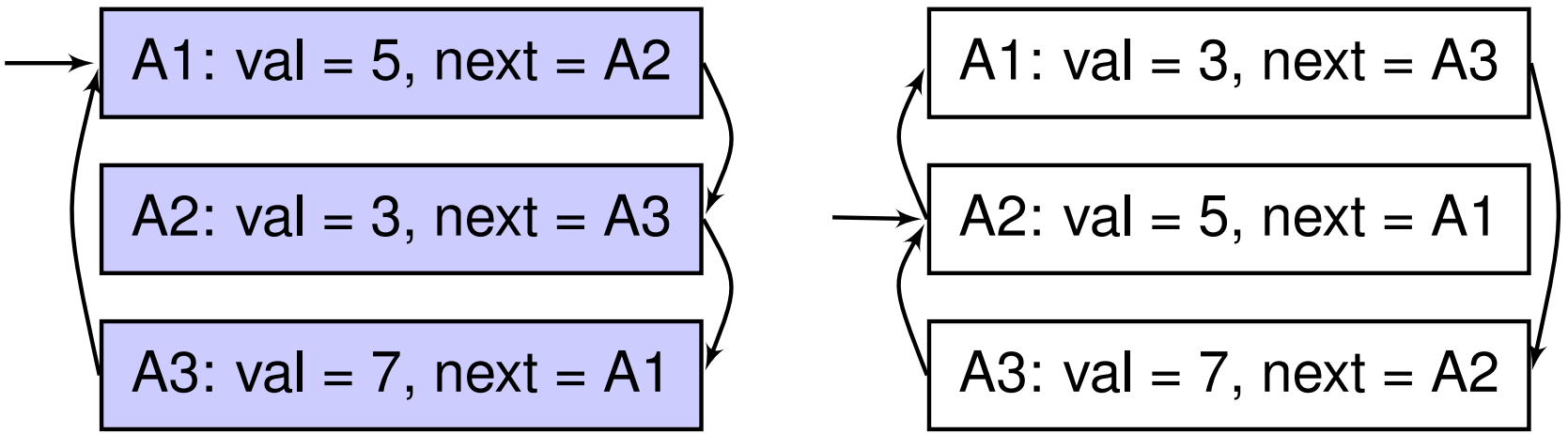


$\rightarrow 5 \rightarrow 3 \rightarrow 7$

- This happens a lot when allocations are interleaved.
- Sort topologically from fixed roots. Keep track of all pointers, instrument code. Get a canonic layout. Magic.

Heap Symmetry

- Sometimes, different states are actually the same.
- $\Rightarrow$ they are symmetric, and we only want one.

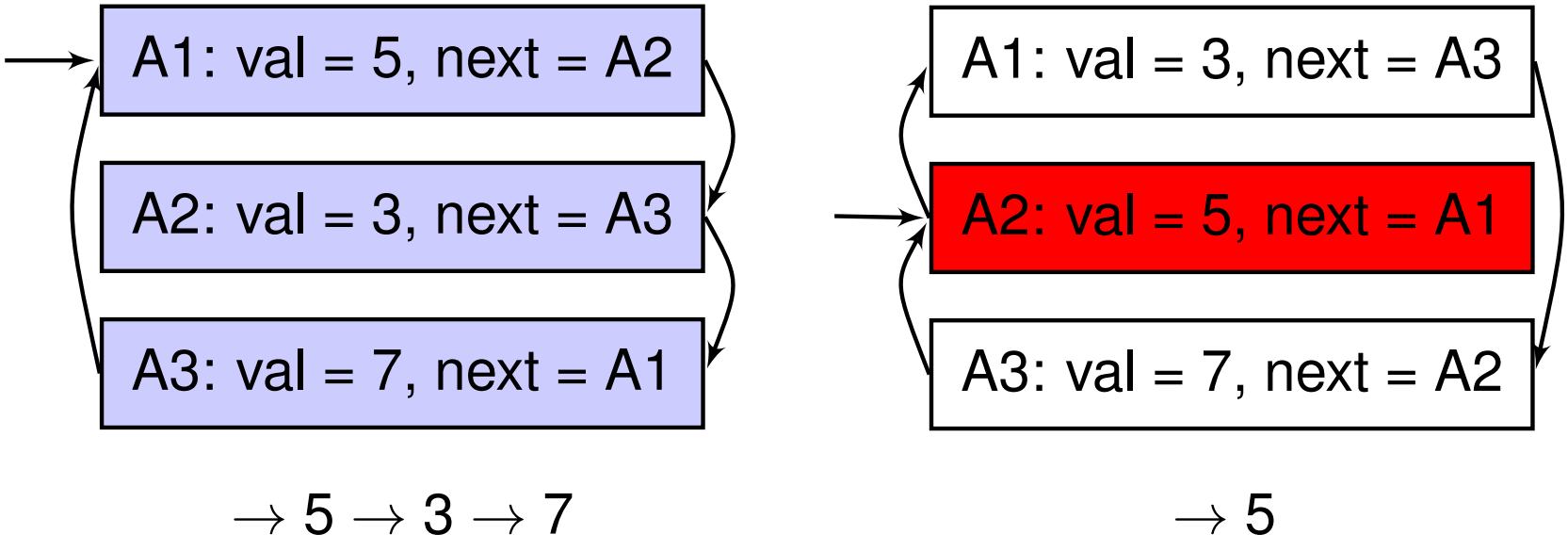


$\rightarrow 5 \rightarrow 3 \rightarrow 7$

- This happens a lot when allocations are interleaved.
- Sort topologically from fixed roots. Keep track of all pointers, instrument code. Get a canonic layout. Magic.

Heap Symmetry

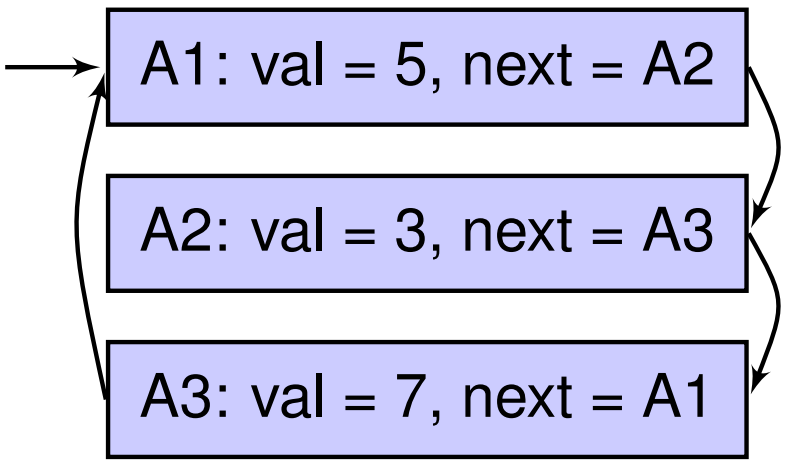
- Sometimes, different states are actually the same.
- $\Rightarrow$ they are symmetric, and we only want one.



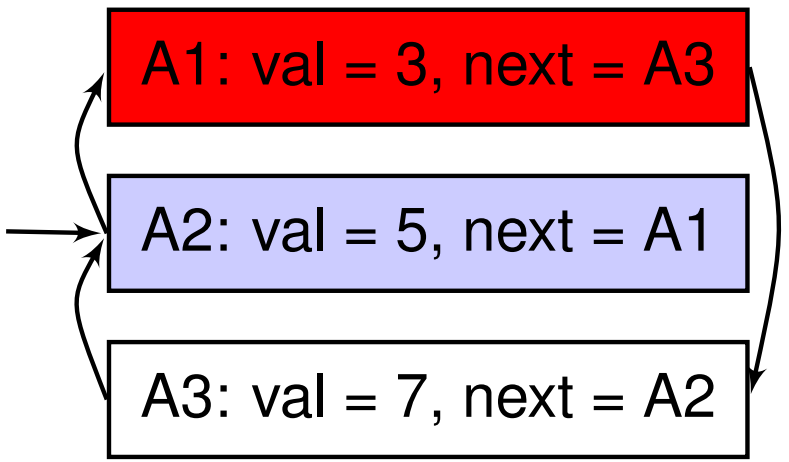
- This happens a lot when allocations are interleaved.
- Sort topologically from fixed roots. Keep track of all pointers, instrument code. Get a canonic layout. Magic.

Heap Symmetry

- Sometimes, different states are actually the same.
- $\Rightarrow$ they are symmetric, and we only want one.



$\rightarrow 5 \rightarrow 3 \rightarrow 7$

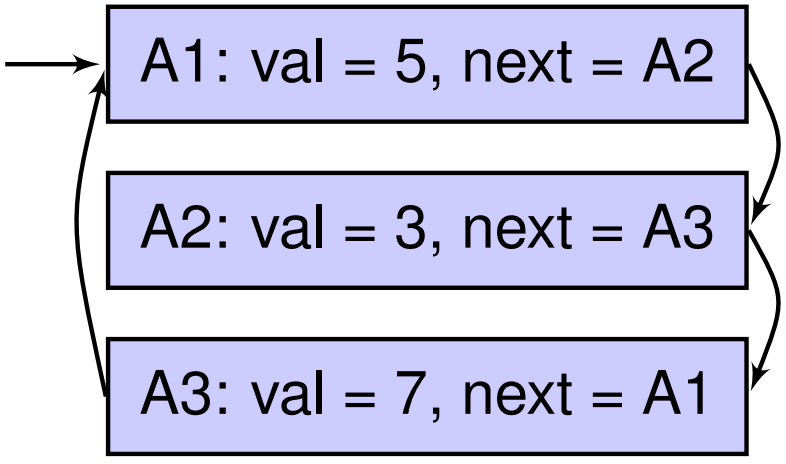


$\rightarrow 5 \rightarrow 3$

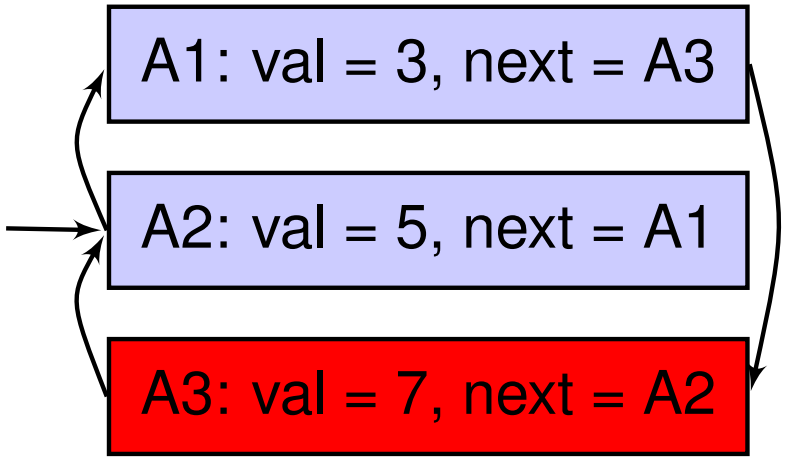
- This happens a lot when allocations are interleaved.
- Sort topologically from fixed roots. Keep track of all pointers, instrument code. Get a canonic layout. Magic.

Heap Symmetry

- Sometimes, different states are actually the same.
- $\Rightarrow$ they are symmetric, and we only want one.



$\rightarrow 5 \rightarrow 3 \rightarrow 7$

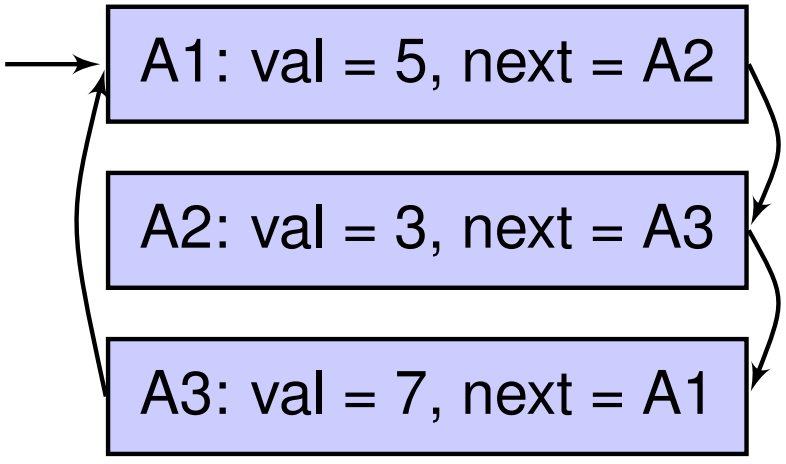


$\rightarrow 5 \rightarrow 3 \rightarrow 7$

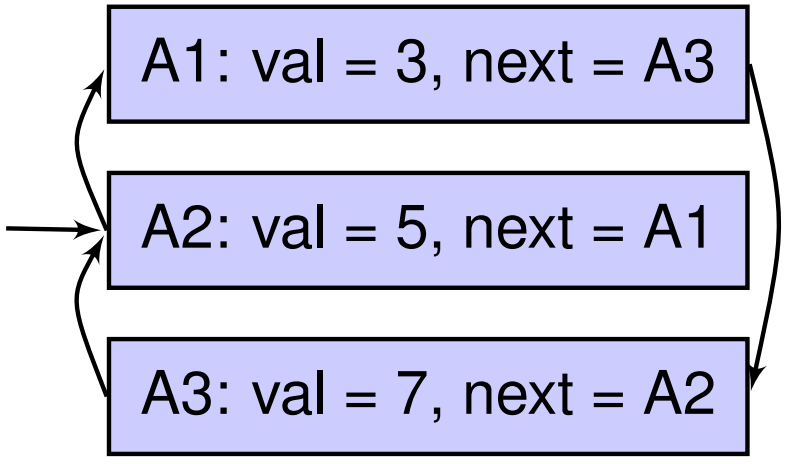
- This happens a lot when allocations are interleaved.
- Sort topologically from fixed roots. Keep track of all pointers, instrument code. Get a canonic layout. Magic.

Heap Symmetry

- Sometimes, different states are actually the same.
- $\Rightarrow$ they are symmetric, and we only want one.



$\rightarrow 5 \rightarrow 3 \rightarrow 7$



$\rightarrow 5 \rightarrow 3 \rightarrow 7$

- This happens a lot when allocations are interleaved.
- Sort topologically from fixed roots. Keep track of all pointers, instrument code. Get a canonic layout. Magic.

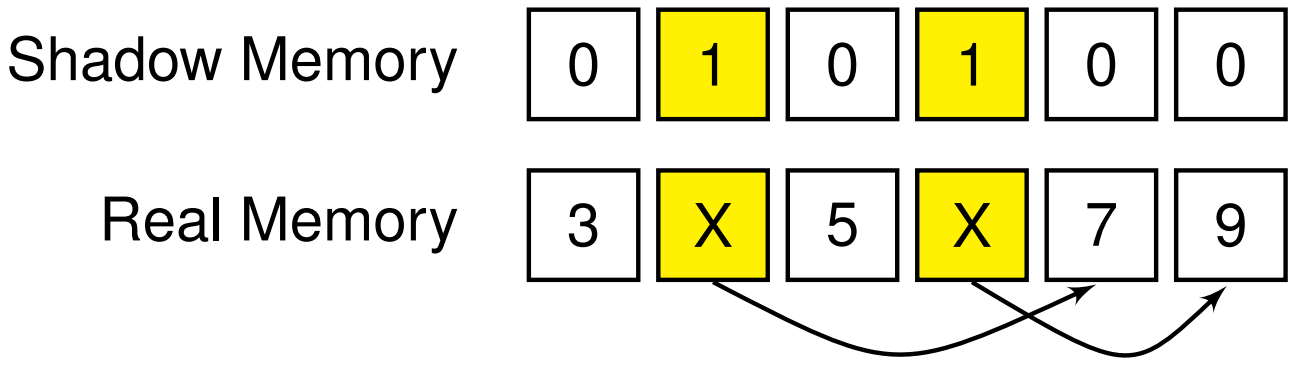
Tracking Pointers

- Low-level languages with type erasure (C, C++),
- with no way to distinguish pointers from other data.

- Magic: **shadow memory** – we instrument all memory and register access with code to keep extra bits of info for each machine word.
- The extra bits include a tag saying “this word is a pointer”.
- Copying, adding to, etc. a pointer results in a pointer.
- Some dubious pointer operations are disallowed (not a problem in practice).

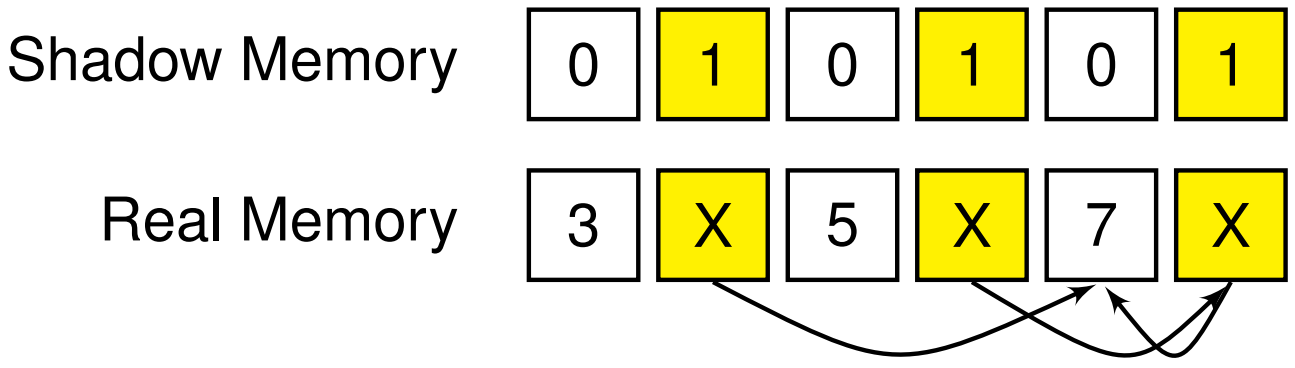
Shadow Memory

Some initial layout:



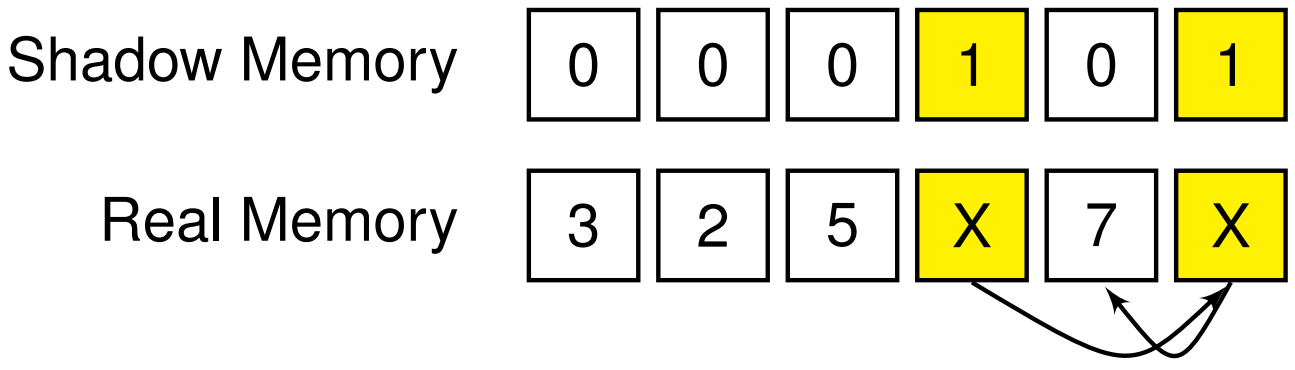
Shadow Memory

Copy memory from address 2 to address 6:



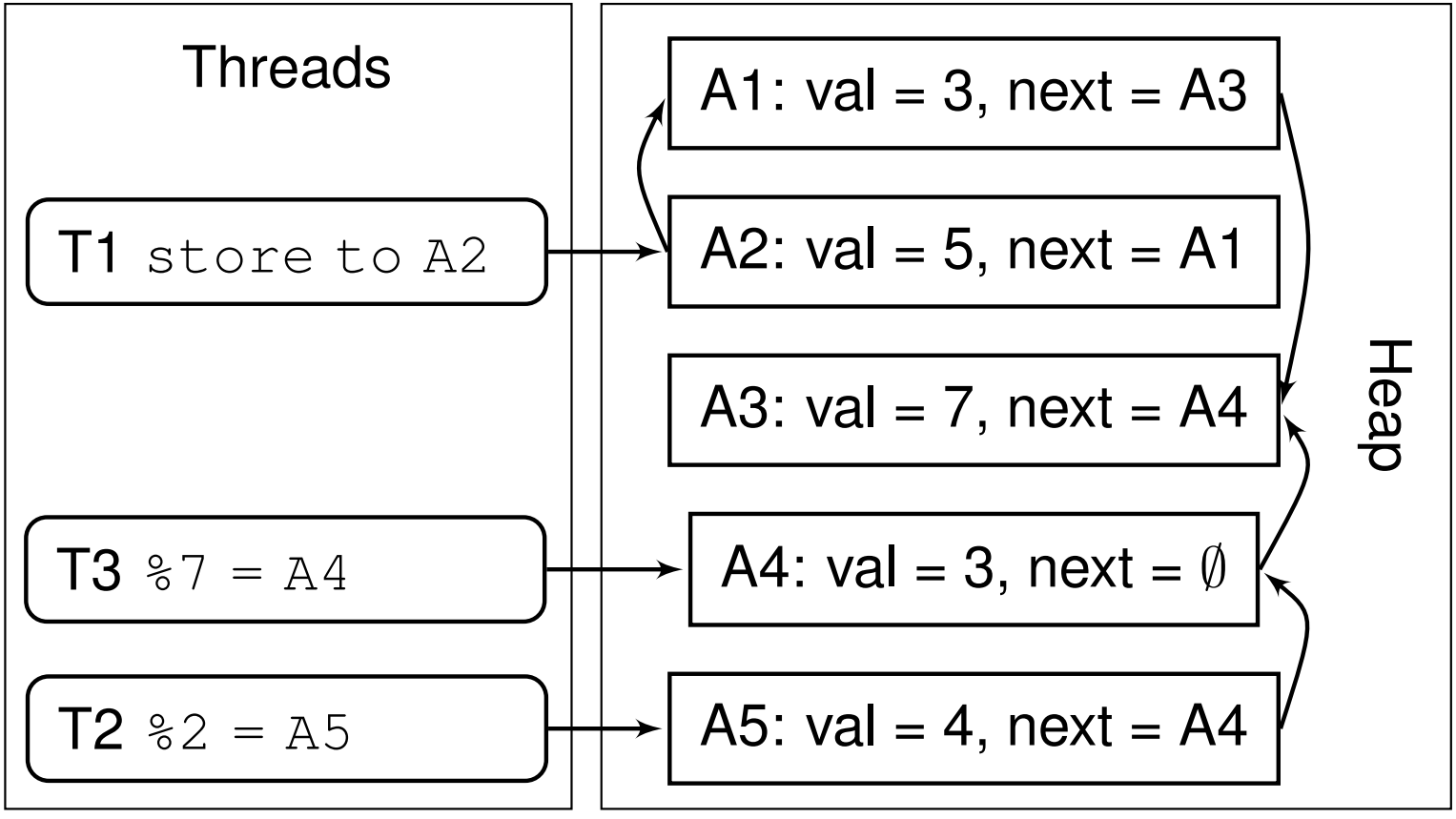
Shadow Memory

Set address 2 to (integral) value 2:



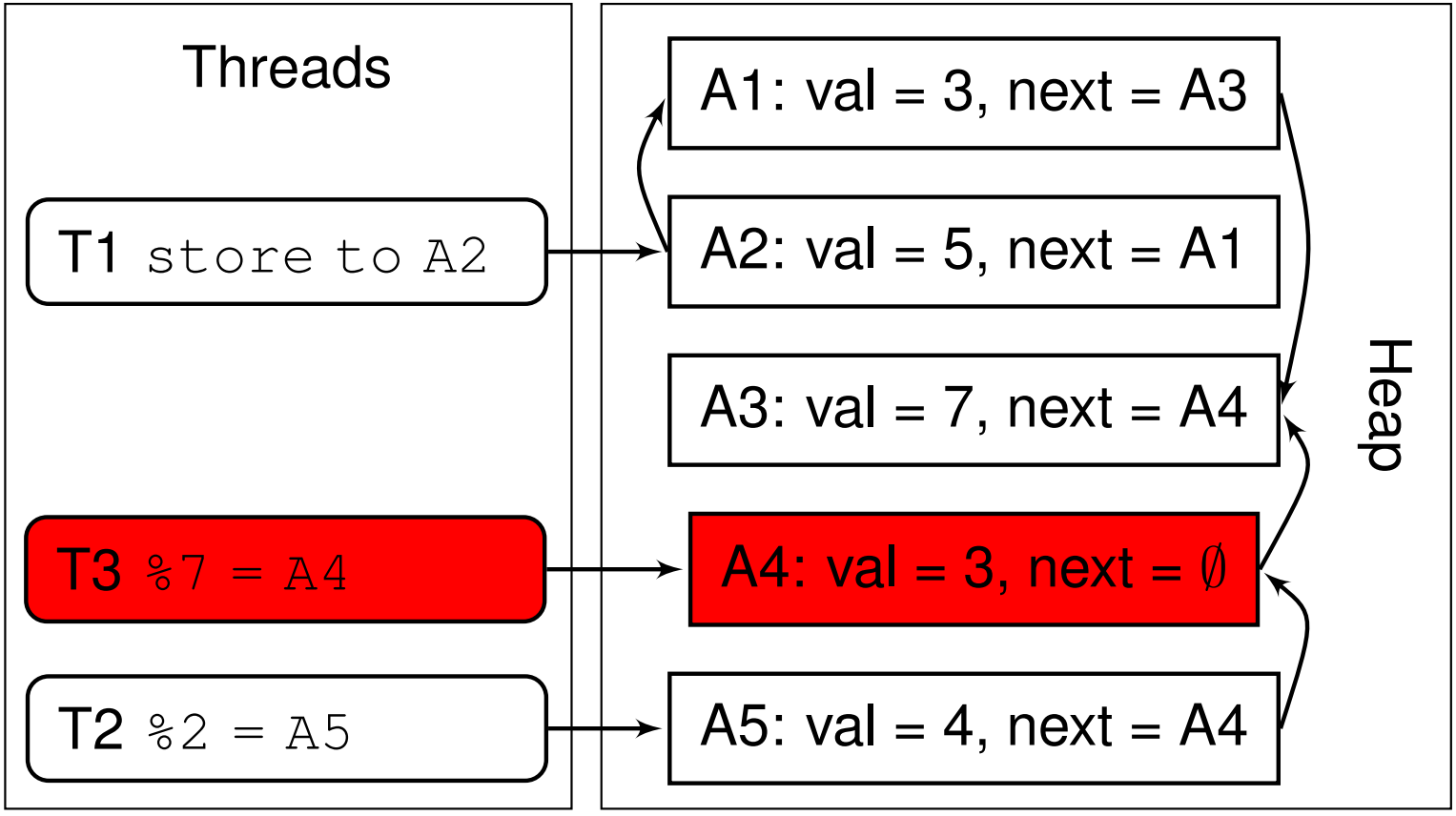
Store Visibility

- $\tau+$reduction treats `store` as a visible operation.
- We can do better than that.



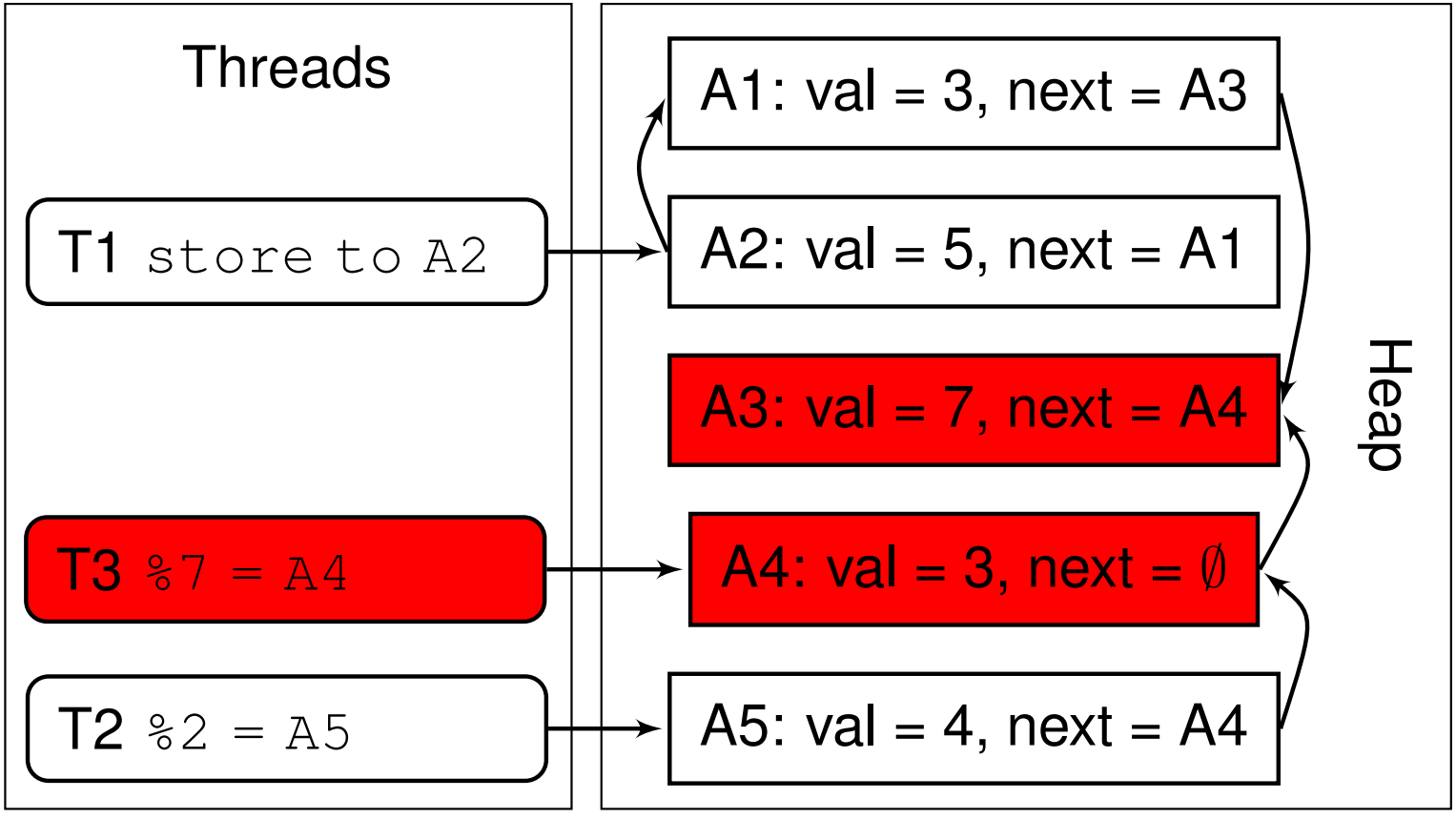
Store Visibility

- $\tau+$reduction treats `store` as a visible operation.
- We can do better than that.



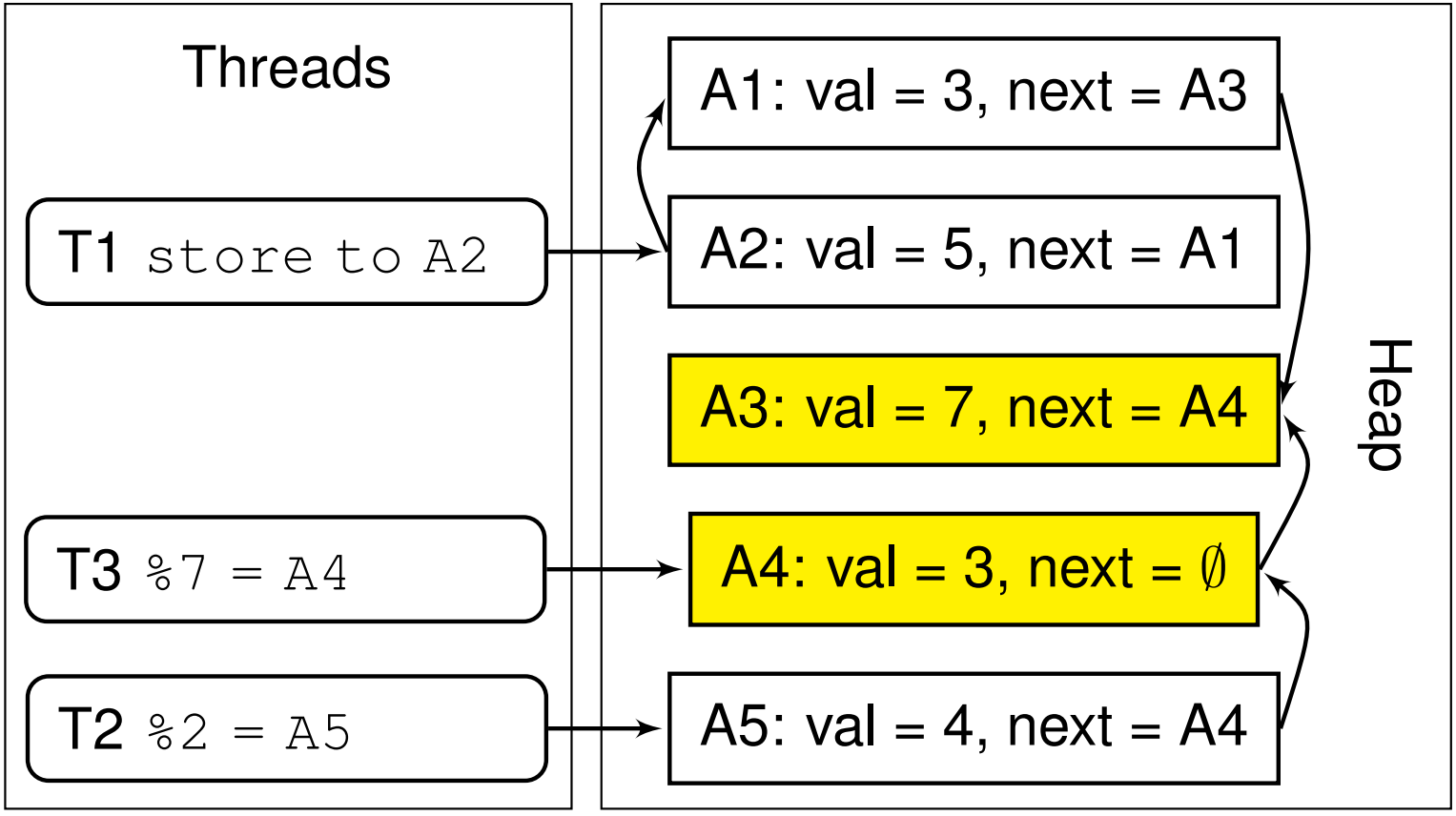
Store Visibility

- $\tau+$reduction treats `store` as a visible operation.
- We can do better than that.



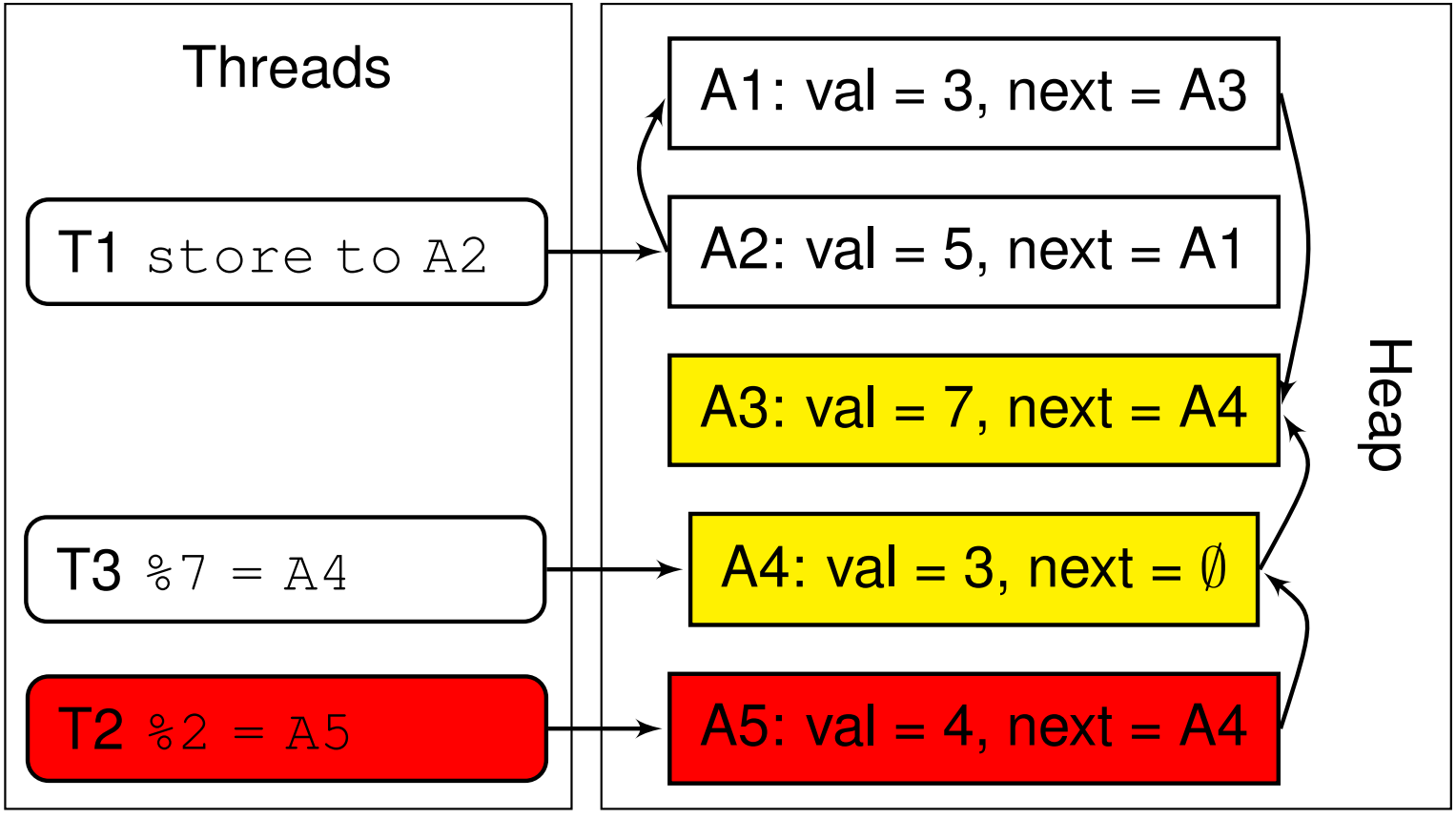
Store Visibility

- τ+reduction treats `store` as a visible operation.
- We can do better than that.



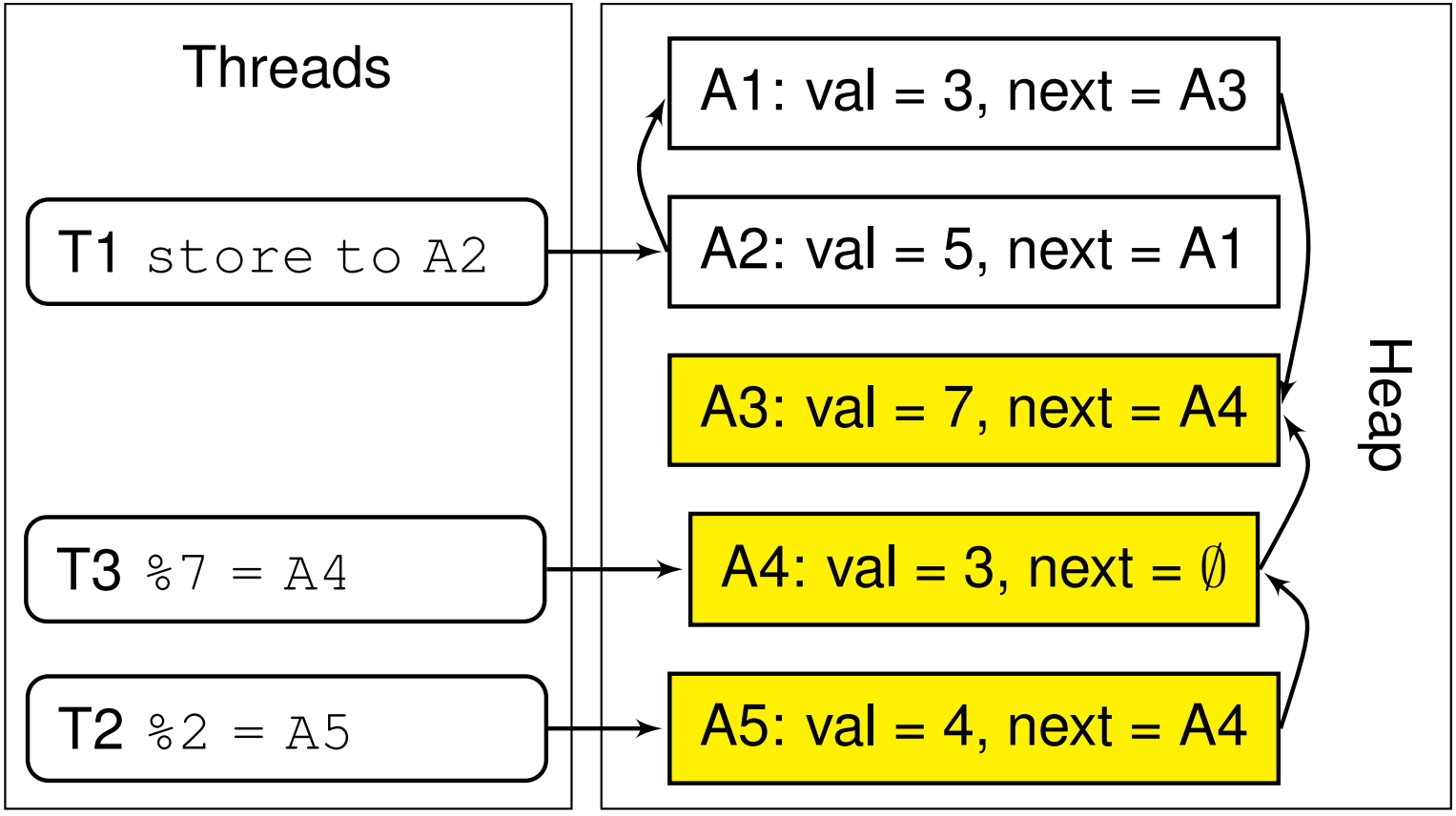
Store Visibility

- $\tau+$reduction treats `store` as a visible operation.
- We can do better than that.



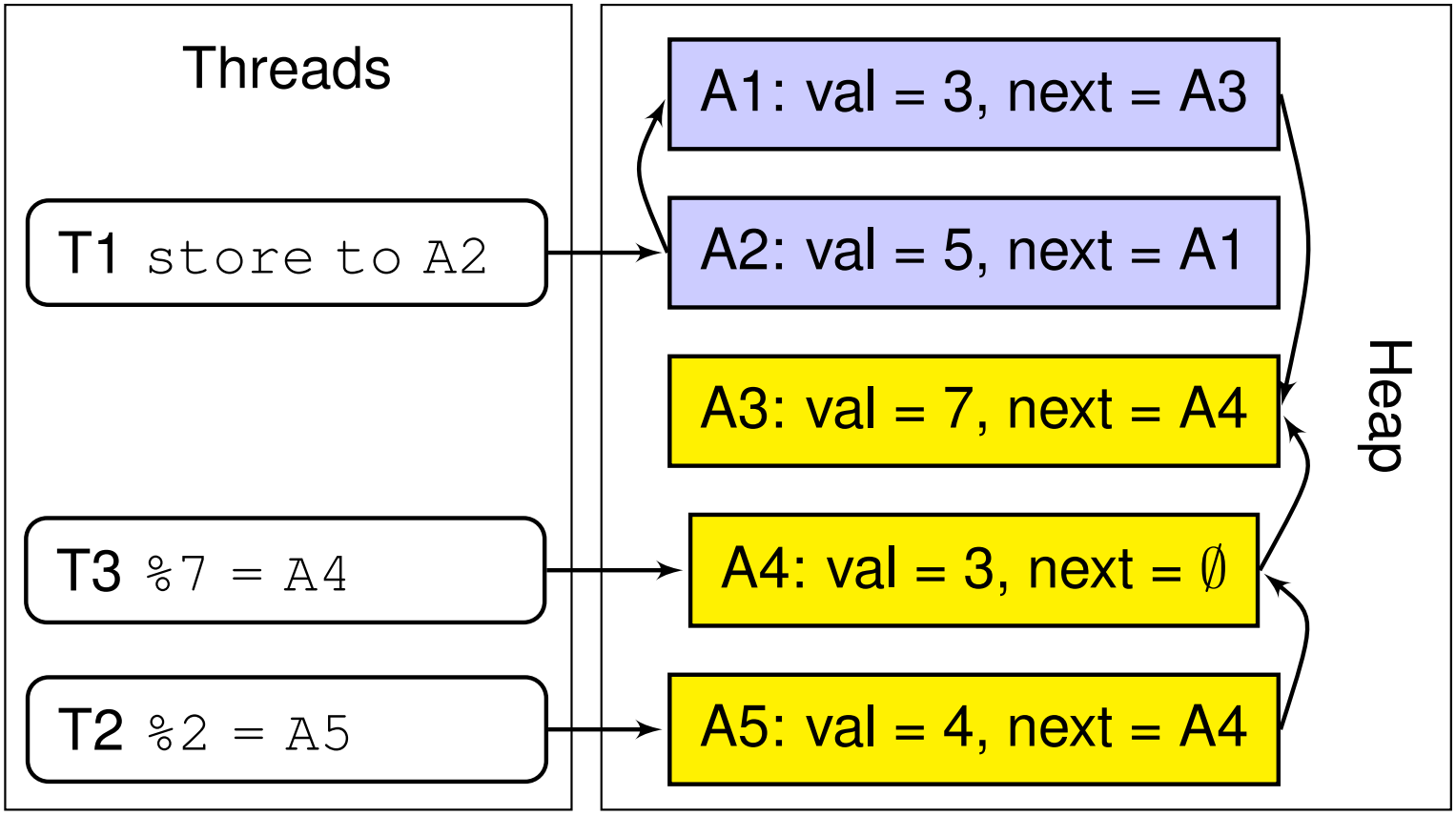
Store Visibility

- τ+reduction treats `store` as a visible operation.
- We can do better than that.



Store Visibility

- τ+reduction treats `store` as a visible operation.
- We can do better than that.



- The store to A2 is **invisible** ,
- as A2 is not reachable from any other thread.

“User-Space” Libraries & Atomicity

- Most of our `pthread.h` implementation is C++ code compiled into LLVM bitcode and linked into your program.
- There is only a handful of real “builtin” functions.

- All this code contributes to state space complexity
- & even worse, it is prone to concurrency bugs.

```
__divine_interrupt_mask();
touch_shared_variables();
global ++; // nobody can see us
__divine_interrupt_unmask();
```

- Saves **a lot** of work, for programmers and DIVINE alike.

Why did the chicken cross the road?

... and was it worth the effort?

Results

A few example models, with `-O0`:

- the infamous peterson: 294193 → 212 states
- a concurrent data structure: 559364 → 108 states

And with `-O2`:

- the infamous peterson: 21122 → 260 states
- a concurrent data structure: 83898 → 143 states

⇒ **590× reduction** for optimized,
5179× reduction for unoptimized bitcode models.

(the red numbers include heap reduction, as it can't be easily disabled)

The Current State of Affairs

- ISO C and most of its standard library work out of the box.
- Most of `pthreads` (POSIX.1c) works out of the box.
- ISO C++11 works partially:
  - no exception support
  - standard library not yet provided in bitcode form
- Programs with a couple threads and small memory footprint can be verified very easily.

Future Work

- Tighter packing of state vectors (better register allocation),
- state space compression,
  - experiments show further ca. 40$\times$ memory use reduction
- better (LTL) property specification.

- ... verification of x86 code?

Grab the code, build it and experiment:

http://divine.fi.muni.cz

You can also **try it online** with some simple C programs!

... please don’t break our server ;-)